A Software Safety Certification Tool for Automatically Generated Guidance, Navigation and Control Code

Ewen Denney
USRA/RIACS
NASA Ames Research Center, Moffett Field, CA 94035
edenney@email.arc.nasa.gov

Steven Trac
University of Miami, Coral Gables, FL 33146
strac@mail.cs.miami.edu

Abstract—Model-based[1,2] design and automated code generation are being used increasingly at NASA. Many NASA projects now use MathWorks Simulink and Real-Time Workshop for at least some of their modeling and code development. The trend is to move beyond simulation and prototyping to actual flight code, particularly in the Guidance, Navigation, and Control domain. However, there are substantial obstacles to more widespread adoption of code generators in such safety-critical domains. Since code generators are typically not qualified, there is no guarantee that their output is correct, and consequently the generated code still needs to be fully tested and certified. Moreover, the regeneration of code can require complete recertification, which offsets many of the advantages of using a generator. Indeed, manual review of autocode can be more challenging than for hand-written code. Since the direct V&V of code generators is too laborious and complicated due to their complex (and often proprietary) nature, we have developed a generator plug-in to support the subsequent certification of the code that is generated. Specifically, the AutoCert tool supports certification by formally verifying that the generated code is free of different safety violations, by constructing an independently verifiable certificate, and by explaining its analysis in a textual form suitable for code reviews. This enables missions to obtain assurance about the safety and reliability of the code without excessive manual V&V effort and, as a consequence, increases the acceptance of code generators in safety-critical contexts. The generation of explicit certificates and textual reports is particularly well-suited to supporting independent V&V. The key technical idea of our approach is to exploit the idiomatic nature of auto-generated code in order to automatically infer logical annotations. These allow the automatic formal verification of the safety properties without requiring access to the internals of the code generator. The approach is independent of the particular generator used but is currently being adapted to code generated using MathWorks Real-Time Workshop, an automatic code generator that translates from Simulink/Stateflow models into embedded C code.

[1] 1-4244-1488-1/08/$25.00 ©2008 IEEE.
[2] IEEEAC paper #1186, Version 4, Updated October 16, 2007

TABLE OF CONTENTS

1. BACKGROUND

We begin in this section with some necessary background on automated code generation and different approaches which can be taken to V&V. We then give the background to our certification approach. In Section 2, we describe our case study with the Vertical Motion Simulator. Next, Section 3 gives the design of the AutoCert plug-in, concentrating on the integration with the Matlab environment. Finally, Section 4 summarizes the tool capabilities and describes our plans for future development.

Automated Code Generation

Model-based design and automated code generation (or autocoding) are being used increasingly at NASA. They promise many benefits, including higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors. There are now numerous successful applications of both in-house custom generators for specific projects, and generic commercial generators. One of the most popular code generators within NASA is the MathWorks Real-Time Workshop [MAT] (with the add-on product Embedded Coder), an automatic code generator that translates Simulink/Stateflow models into embeddable (and embedded) C code. By some estimates, 50% of all NASA projects now use Simulink and Real-Time Workshop for at least some of their code development. Code generators have traditionally been used for rapid prototyping and design exploration, or the generation of certain kinds of code (user interfaces, stubs, header files etc.), but there is a clear trend now to move beyond simulation and prototyping to the generation of production flight code, particularly in the Guidance,

Navigation, and Control domain. Indeed, the prime contractor for the Orion Spacecraft (NASA's Crew Exploration Vehicle) is making extensive use of code generators for the development of the flight software.

Nevertheless, there remain substantial obstacles to more widespread adoption of code generators in such safety-critical domains, principally, how the generated code should be assured. Ideally, the code generator, itself, should be qualified. However, this is a non-trivial and expensive process, and is therefore rarely done. Moreover, the qualification is only specific to the use of the generator within a given project, and needs to be redone for every project and for every version of the tool. Also, even if a code generator is generally trusted, user-specific modifications and configurations necessitate that V&V be carried out on the generated code [Erk04]. Since code generators are typically not qualified, there is no guarantee that their output is correct, and consequently the generated code still needs to be fully tested and certified.

There are generally two workarounds for dealing with code generator bugs. Sometimes there is a model workaround - i.e., modify the model. This will likely not always be an option. Moreover, some bugs can not be easily characterized at the model level - that is, it is difficult to say which combinations of model elements give rise to these bugs, let alone how to fix the models.

The second option is simply to upgrade to a newer version of the generator. However, any qualification effort which has been carried out on the previous working version is now lost, the code must be recertified, and the entire toolchain must now essentially be upgraded. This can offset many of the advantages of using a generator.

Moreover, advocates of the model-driven development paradigm claim that by only needing to maintain models, and not code, the overall complexity of software development is reduced. While it is undoubtedly true that some of burden of verification can be raised from code to model, it should be acknowledged that, in fact, there are additional concerns and, indeed, more artifacts in a model-based development process. Users need to be sure that the code implements the model, that the code generator is correctly used and configured, that the target adaptations are correct, that the generated code meets high-level safety requirements, that it is integrated with legacy code, and so on. There can also be concerns with the understandability of the generated code. Some understanding of *why* the code is safe, therefore, helps the larger certification process. Automated support for V&V that is integrated with the generator can address some of these complexity concerns.

Furthermore, certification requires more than black box verification of selected properties, otherwise trust in one tool (the generator) is simply replaced with trust in another (the verifier). Finally, the direct V&V of code generators is too laborious and complicated due to their complex (and often proprietary) nature, while testing the generator itself can require detailed knowledge of the transformations it applies [SC03, SWC05].

Automated code generation, therefore, presents a number of challenges to software processes and, in particular, to V&V, and this leads to risk. The AutoCert tool we describe here mitigates some of that risk.

Automated Code Certification

In contrast to approaches based on directly qualifying the generator, itself, or on testing of the generated code, we instead propose a generator plug-in to support the subsequent certification of the code created by the generator. Specifically, our tool will support certification by formally verifying that the generated code is free of different safety violations, by constructing an independently verifiable certificate, and by explaining its analysis in a textual form suitable for code reviews. This will enable missions to obtain assurance about the safety and reliability of the code without excessive manual V&V effort and, as a consequence, increase the acceptance of code generators in safety-critical contexts. The generation of explicit certificates is particularly well-suited to supporting independent V&V. The key technical idea of our approach is to exploit the idiomatic nature of auto-generated code in order to automatically infer logical annotations. Annotations are crucial in order to allow the automatic formal verification of the safety properties without requiring access to the internals of the code generator, as well as making a precise analysis possible. The approach is independent of the particular generator used, and need only be customized by the appropriate set of patterns.

Now, considering the case where no bugs are detected, it is guaranteed that the auto-generated source code is free of violations, and we can compare the time taken to review and certify the auto-generated code by hand, with the time taken to do it with support from AutoCert. This support consists of automatically checking that the code complies with the specified safety properties, generating an explanation for why it complies, and tracing this explanation to code, model, and verification artifacts.

Real-Time Workshop has extensive tracing capabilities. However, optimization can obscure connections between model blocks and corresponding code fragments, by merging and compressing functionally separate fragments. As part of its analysis, AutoCert "reverse engineers" the code, sifting through potentially overlapping fragments to create links from the code to high-level functional descriptions (in an auto-generated safety document).

We follow the tradition in formal methods of referring to techniques which conclusively demonstrate the absence of bugs (rather than simply search for existing bugs) as

performing certification. However, in an IV&V context, we must consider the larger picture of certification, of which formal verification is a part, and therefore produce assurance evidence which can be checked either by machines (during proof checking) or by humans (during code reviews).

Rather than use a separate third-party analysis tool, we are designing a plug-in that is tightly coupled to the Real-Time Workshop code generator. We adopt the title, AutoCert/RTW (AutoCert for short), for this safety certification plug-in. Following the plug-in philosophy, the tool acts as an extension of RTW, and is therefore closely integrated from the user's perspective, but the implementation does not require a deep integration with the internal operations of RTW.

The following sections describe the components of our system: the style of safety properties which we check, the inference of annotations, the creation and discharge of verification conditions, the generation of safety documents, and the overall system architecture.

Safety Properties—AutoCert supports certification by formally verifying that the generated code is free of violations of specific safety properties. In our approach, we distinguish between various kinds of safety properties. Language-specific properties concern those safety aspects of the code which only depend upon the semantics of the programming language. Examples include memory safety (e.g., absence of array bounds violations), variable initialization, and scoping requirements. Domain-specific properties relate to details which are specific to the use of a given code generator in a particular domain. For example, all values of x for an interpolation table (x,y) must be disjoint and in increasing order. Finally, project-specific and application-specific properties talk about guarantees for a family of applications or a single application, respectively. For example, flight-rules can be considered to comprise typical project-specific properties.

A range of safety properties, including initialization safety, and absence of out-of-bounds array accesses, have already been formalized and can be used with our algorithm. Initialization safety ensures that each variable or individual array element has been explicitly assigned a value before it is used. Array-bounds safety requires each access to an array element to be within the specified upper and lower bounds of the array, and is a typical example of a language-specific property. Matrix symmetry requires certain two-dimensional arrays to be symmetric. Sensor input usage is a GN&C specific property which is a variation of the general init-property guaranteeing that each sensor reading passed as an input to a state estimation algorithm is actually used during the computation of the output estimate. Frame safety checks that each variable is in the correct coordinate frame, and that coordinate transformation are correctly applied. Another example, from the data analysis domain, ensures that certain one-dimensional arrays represent normalized vectors, i.e., that their contents add up to one. Details of how safety properties are formalized in our approach are omitted here.

Hoare-Style Safety Certification—Our certification approach uses the well-known Hoare-style framework to prove the safety properties. This is based on proof rules that derive triples of the form P {C} Q, meaning "if pre-condition P holds before execution of statement C, then Q holds after". For each safety property and each statement a corresponding rule is given. A verification condition generator (VCG) then applies the rules to a program, which produces a number of logical statements or proof obligations. Unfortunately, the Hoare-style framework requires a large amount of logical annotations attached to statements of the code, which describe pre- and post-conditions and loop invariants. This has so far limited its practical applicability. However, it is important to observe that correctness of the proofs does not depend on correctness of the (untrusted) annotations; rather, they can be seen as hints which guide the proof process. This allows us to automatically infer the annotations without compromising the safety guarantees provided by the certification tool.

For each notion of safety the appropriate safety property and corresponding policy must be formulated. This is usually straightforward; in particular, the safety policy can be constructed systematically by instantiating a generic rule set that is derived from the standard rules of the Hoare calculus [DF03]. The basic idea is to extend the standard environment of program variables with a "shadow" environment of safety variables which record safety information related to the corresponding program variables. The rules are then responsible for maintaining this environment and producing the appropriate verification conditions (VCs). Safety certification then starts with the outermost (i.e., at the end of the program) postcondition true and computes the weakest safety precondition (WSPC), i.e., the WPC together with all applied safety conditions and safety substitutions. If the program is safe then its WSPC will be provable without any assumptions.

Annotation Inference—For arbitrary (i.e., manually written) code it is impossible to automatically generate the required annotations and most annotations must be provided by the user—a prohibitively tedious and costly task. However, a code generator like RTW produces highly structured and idiomatic code. Consequently, only a few, standardized annotations need be used. Intuitively, *idiomatic code* exhibits some regular structure beyond the syntax of the programming language and uses similar constructions for similar problems. Even manually written code already tends to be idiomatic, but the idioms used vary with the programmer, and are much less regular. Automated generators eliminate this variability because they derive code by combining a finite number of building blocks.

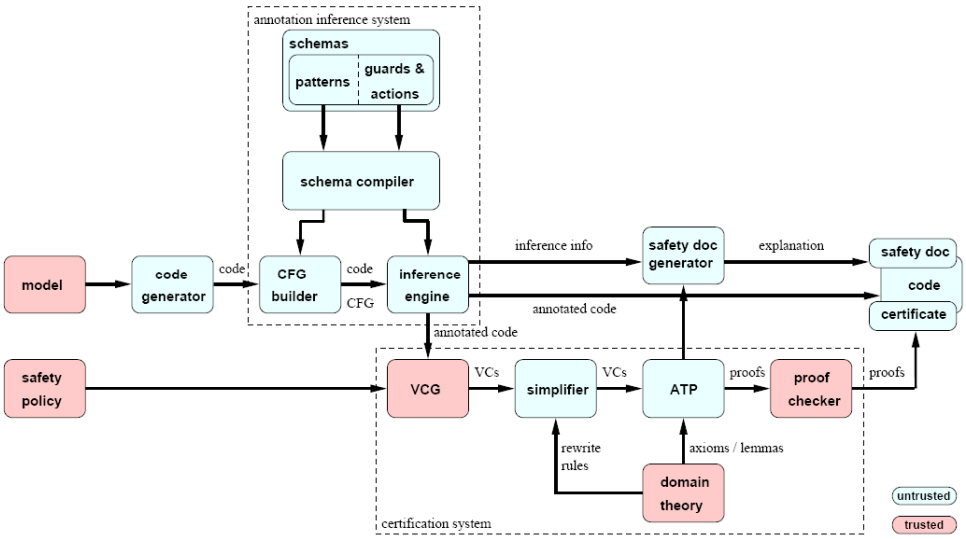

Figure 1 – AutoCert: System Architecture

The idioms determine the interface between the code generator and the inference algorithm. For each generator and safety property, our approach thus requires a customization step in which the relevant idioms are identified and formalized as patterns. Note that neither missed idioms nor wrong patterns can compromise the assurance given by the safety proofs because the inferred annotations remain untrusted. They can, however, compromise the "completeness" of the approach in the sense that safe programs can fail to be proven safe, and in our experience, a few iterations can be required to identify all patterns. Note also that the idioms can be recognized from a given code base alone, even without knowing the templates that produced the code. This gives us two additional benefits. First, it allows us to apply our technique to black-box generators. Second, it also allows us to handle optimizations: as long as the resulting code can still be characterized by patterns, neither the specific optimizations nor their order matter.

We have developed a generic *pattern language* to describe these code idioms. The patterns let us define *annotation schemas* to encapsulate certification cases for matching code fragments. We omit details of the schema language here. An *annotation schema compiler* takes a collection of annotation schemas tailored towards a specific code generator and safety property, and compiles it down into a customized annotation inference algorithm. The annotation schemas are then applied using a combination of planning and aspect-oriented techniques to produce an annotated program, which can then be certified in the Hoare-style framework. We can thus check conformance of generated code with a range of safety properties fully automatically. As an example, consider a matrix that is initialized using a nested loop. In order to verify that the code completely initializes the matrix, we need at least four annotations: inner and outer loop invariants, which formalize "snapshots" of the matrix initialized "up to that point", and inner and outer post-conditions, which formalize successful initialization of all or part of the matrix. Different annotations are required for row-major and column-major memory layouts. Additional complications arise when information from the initialization block needs to be propagated to parts of the code where it is needed, taking into account scope, control flow, and context. However, although the resulting annotations can become quite complex, several underlying principles can be used to generate them automatically. The only input which is needed is the basic pattern of two-dimensional iteration (which captures both memory layouts), and a definition of the initialization safety property. We have a library of schemas which allows us to certify code generated by RTW from a range of models, as well as using in-house code generators. We have also used the engine to analyze code produced from models in the VMS project (Section 2).

VC Processing—A Verification Condition Generator (VCG) traverses the annotated code and applies the rules of the calculus to produce Verification Conditions (VCs). These are logical formulas which need to be shown to ensure compliance with the safety property. The VCG simply implements the semantics of the programming language and the proof rules of the safety policies. The VCs are then simplified, completed by an axiomatization of a background theory and given as proof obligations to an off-the-shelf high-performance automated theorem prover (ATP). If all obligations are proven it is guaranteed that the safety property is obeyed and the resulting proofs comprise the evidence for that. The VCG can be seen, therefore, as performing a *compositional* verification of the property.

We use automated theorem proving to check the VCs. In contrast to forms of theorem proving which are interactive (mainly tactic-based higher-order provers), we use customized domain theories of logical axioms, and scripts, so that the prover is essentially used as a decision procedure, and its use is completely hidden from the user. We use the

TPTP syntax [TPTP] which lets us use a wide range of the off-the-shelf first-order provers.

Safety Documentation—Rather than act as a black-box verification tool which provides a simple pass/fail result, AutoCert provides a detailed *safety documentation report*. The report is generated from the analysis of the code and provides a high-level traceable explanation of *why* the code complies with the specified safety property. The report is intended to help users in understanding the generated code (often a particular concern for automatically generated code, and to support the manual process of code review. Also, by explaining the reasoning behind the certification process, there is less of a need to trust the tool. The report can draw attention to potential certification problems.

If we suppose that a diligent code reviewer must "rediscover" all the information which is automatically generating by AutoCert, in order to construct a watertight justification of safety, even for a small program this can result in substantial savings in effort.

The report first lists all the "relevant" variables. Intuitively, they are the variables to which reviewers are likely to need to direct their attention; technically, these are variables for which the logical proof of safety passed a certain threshold of complexity. This and other features could be further customized using style templates.

Then, for each variable in turn, the report explains why the variable meets the requirement. The explanation can contain explanations of fragments of code, which can lead to explanations for other variables (which are cross-linked). Whenever the tool carries out some analysis using the prover (e.g., that a code fragment establishes some property), it provides links to the corresponding verification conditions.

System Architecture—Figure 1 shows the overall system architecture of our certification approach. At its core is the original (unmodified) code generator (in this case, Real-Time Workshop) which is complemented by the annotation inference subsystem, including the pattern library and the annotation schemas, as well as the standard machinery for Hoare-style techniques, i.e., VCG, simplifier, ATP, domain theory, and proof checker. The analysis proceeds by first translating the parsed C code into a simplified intermediate language. The logical inference is carried out on this language. The inference engine also supplies information to the safety document generator, which renders this along with the code. The architecture distinguishes between trusted and untrusted components, shown in Figure 1 in red (dark grey) and blue (light grey), respectively. *Trusted* components *must be correct* because any errors in them can compromise the assurance provided by the overall system. *Untrusted* components, on the other hand, are not crucial to the assurance because their results are double-checked by at least one trusted component. In particular, the assurance provided by our approach does not depend on the correctness of the two largest (and most complicated) components: the original code generator, and the ATP; instead, we need only trust the safety policy, the VCG, the domain theory, and the proof checker. Moreover, the annotation inference subsystem (including the pattern library and annotation schemas) also remain untrusted since the resulting annotations simply serve as "hints" for the subsequent analysis steps. We will omit further technical details. These components and their interactions are described in more detail in publications [DF03, DFS06, 4, 5].

2. CASE STUDY: VERTICAL MOTION SIMULATOR

We have applied AutoCert to analyze code which has been autogenerated using RTW, from Simulink models provided by the Vertical Motion Simulator (VMS) facility at NASA Ames.

Overview of the VMS

The Vertical Motion Simulator (VMS) is a world-class research and development facility located in the Aviation Systems Division at NASA Ames Research Center that offers unparalleled capabilities for conducting experiments involving aeronautics and aerospace disciplines. The six-degree-of-freedom VMS, with its 60-foot vertical and 40-foot lateral motion capability, is the world's largest motion-base simulator. The large amplitude motion system of the VMS was designed to aid in the study of helicopter and vertical/short take-off landing (V/STOL) issues specifically relating to research in controls, guidance, displays, automation, and handling qualities of existing or proposed aircraft. It is also an excellent tool for investigating issues relevant to nap-of-the-earth flight, and landing and rollout studies.

Since the VMS is effectively a piloted vehicle, the system must be human rated. Specifically, the VMS satisfies NPR 8705.2A, "Human Rating Requirements for Space Systems".

Mode Control Unit

The VMS has four hydraulic axes. Three rotational axes control roll, pitch, and yaw, respectively, and a linear axis that controls longitudinal movements. The VMS developers provided a Simulink block diagram of a single hydraulic rotational servo axis controller for use with our analysis tool. Although the Simulink block diagram provided has not yet been implemented into the VMS system, plans are underway to replace the old analog electronics that now deliver this functionality. This model was originally built with MATRIXx System Build block diagrams. Preliminary testing was conducted with this model controlling the simulator motion. Later, the model was manually converted from MATRIXx to Simulink and it is this Simulink model that will be integrated into the VMS.

The hydraulic axis model will be executed on a VME platform with a Motorola single board computer. VxWorks will be used as the real-time operating system. Real-Time Workshop will be used to generate C code from the Simulink model. Analog and discrete, input and outputs are provided by third party vendor VME boards. The model implements a servo loop controller with a servo current loop, a velocity loop and a position loop. The model accepts position and velocity feed forward signals over a fiber optic digital network and provides current drive to the hydraulic actuator. Another controller in the VMS is the Mode Control Unit (MCU) which provides the interface between the host aeronautic computer and the motion control electronics and provides manual control for the motion safety operator. This unit, once implemented with analog electronics was replaced by a digital controller built up on VME using MATRIXx and its components, System Build, AutoCode and RealSim. Plans are in place to convert this system to Simulink by manually converting the model and then using Real-Time Workshop to produce C code that will run under VxWorks.

IV&V

Since the VMS project is moving to the use of a new autocoder, namely Real-Time Workshop, the engineers are interested in tools which can ease the transition from the previous MATRIXx models. The VMS team supplied us with their Simulink model for the MCU, and described the settings they typically use for generating code using RTW. After confirming that we were able to generate the same C code as the VMS team, we analyzed the code for the initialization safety property using a range of analysis settings.

On most settings, the code could be verified with all VCs already discharged (i.e., proven) by the simplifier. This takes under one minute. At the other extreme, performing no simplification at all produced over 700 VCs. Some experimentation was required to determine the settings which provided the most insightful output.

V&V activities for the conversions to digital controllers are done in the VMS at the system level. This is a time consuming process but is critical to get safety certification for human occupancy. For the conversion of the MCU from MATRIXx to Simulink, the VMS team is replacing only the software on the device and the same hardware platform will be used. The VMS developers report that they expect to see much benefit from a tool that would help them to verify that the new software (autocoded by RTW) behaves like the old software, adhering to all the requirements, without having to repeat all the many functional tests that were required when the initial installation of digital equipment was done. AutoCert can obviate the need to construct a huge test-suite to ensure that no low-level errors exist, and therefore helps engineers concentrate on higher-level properties.

3. INTEGRATION WITH MATLAB ENVIRONMENT

In this section we describe the integration of the annotation inference engine with the Matlab environment, in general, and Real-Time Workshop, in particular. There are several ways in which this could be done: using the Matlab command line, using the RTW configuration capabilities, and through the Simulink graphical environment. We first describe the Matlab environment, then present a number of use cases for the functionality which we support. Then we describe the implementation approach which we have adopted, which centers on generating JavaScript from the certification artifacts, and weaving this with RTW-generated files, in order to produce browsable verification artifacts. We also discuss some alternative implementation strategies.

Matlab Environment

Simulink is the MathWorks environment for creating graphical models of dynamic systems (Figure 2). Real-Time Workshop is not a standalone tool, but rather a set of menu options within Simulink, which allow executable C and C++ code to be generated from a model. A further add-on product, Embedded Coder (EC) has various additional features which are useful for generating C code tuned for embedded devices. RTW provides browsing capabilities for its generated code by generating parallel HTML files, which can be viewed with the Matlab Web Browser. That is, for every .c and .h file it generates a parallel c.html and h.html file. The parallel HTML files created by RTW contain internal hyperlinks to type declarations and external hyperlinks back to corresponding Simulink model elements. The Matlab Browser supports a protocol which allows Matlab commands to be invoked from hyperlinks within HTML documents. This is possible since the Matlab Browser resides within the Matlab environment. Thus, clicking on a hyperlink to a model element in the parallel HTML file (inside the Matlab Browser) causes the corresponding box in the Simulink model to be highlighted. Note that like any web browser, the Matlab Browser might not support all of the HTML or related features used in a particular web site or HTML page.

We integrate AutoCert with RTW as follows. We interweave the RTW HTML pages with the annotations obtained from the annotation inference over the parallel C code generated by RTW. This gives the annotated RTW HTML page the ability to link to the model. Since it is possible to invoke other Matlab commands inside the Matlab Browser, AutoCert is able to make Matlab command line calls in order to invoke system calls which can, in turn, again execute the inference engine.

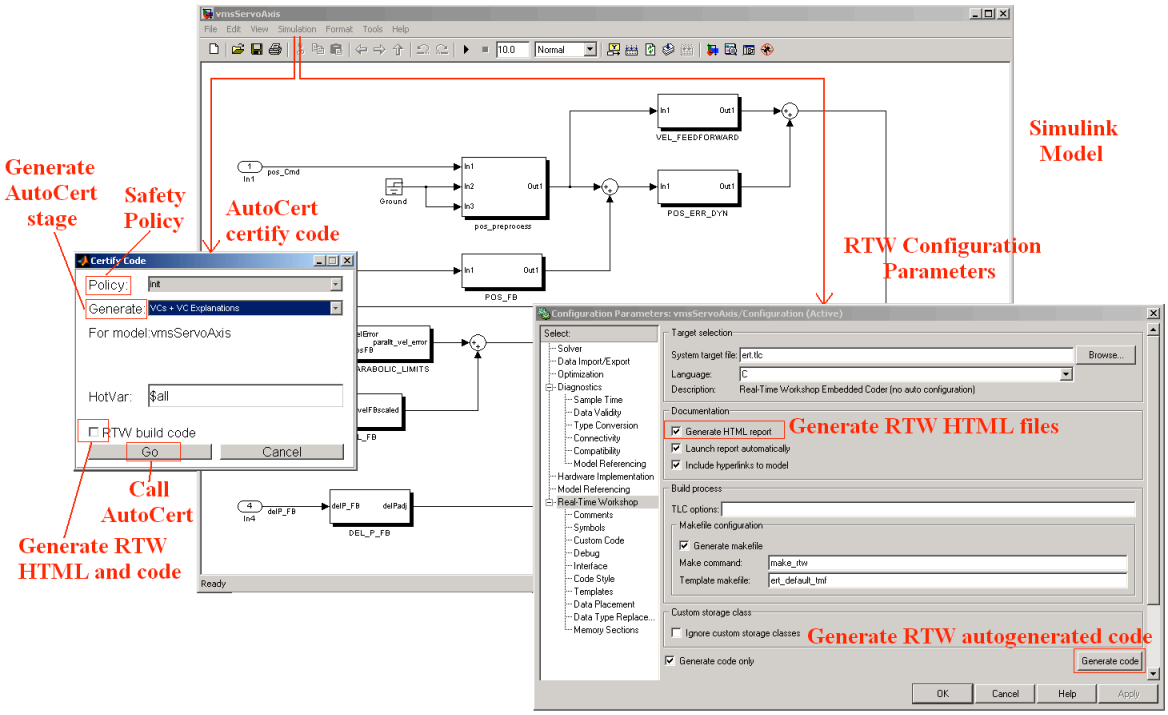

Figure 2 – AutoCert: Certification Menu within Simulink

Certification Functionality

We describe the integration of AutoCert with RTW from the point of view of the user via a series of use cases or scenarios. The integrated tool provides functionality in three main areas: code creation, certificate and safety document creation, and tracing between the various artifacts.

Code Creation—The user has two ways of generating code from their model using RTW. The first is the standard way by using the RTW menu options inside Simulink. Inside the menu options, located under Simulation → Configuration Parameters → Real-Time Workshop, there is a "Generate code" button which creates the code. There are numerous other parameters that the user can use to tune model and target configurations. One of the parameters needed for AutoCert purposes is the option to generate parallel HTML files; this parameter is highlighted in Figure 2.

The second way to generate the code is to directly use the AutoCert menu option, added as a Simulink menu option at Simulation → Certify Code. The "RTW build code" check box option in this menu uses the default settings, which are set inside the RTW Configuration Parameters. When this option is checked off, the code is created, along with the RTW HTML files. The check box option provides two benefits to the user. First, the user can auto-generate code, create RTW HTML files, and call AutoCert all in one menu. Second, the user has the option to autogenerate code before or during the use of the AutoCert menu. The user isn't required to always have code and HTML files pre-generated, allowing some flexibility. The remaining parameters in the AutoCert menu relate to certificate creation, described next.

Certificate Creation—It is assumed that the user has either already generated code from their model using RTW, or that otherwise the "RTW build code" option is checked off in the AutoCert menu. Inside the menu, there are two parameters: select policy and select stage. In order to invoke the inference engine, the user has to select a safety policy from a predetermined list of choices - this choice determines which property of the code is to be certified. AutoCert will then certify the code via a number of stages. The user can also specify that a specific variable ("hotvar") be analyzed with respect to the safety property, or that all variables be checked. Next the user selects a specific certification stage. The possible stages are to generate annotations, VCs, or certificates. There is also an option to generate a safety document.

When "Go" is selected in the AutoCert Certify Code menu, the inference engine is executed (via the Matlab command line) with the options chosen by the user. The RTW generated code and HTML files are passed in as input. The tool then outputs AutoCert HTML pages based on the stage chosen. After finishing outputting the HTML files, the AutoCert menu then opens the Matlab Browser using the Matlab web command, and displays the main certification HTML page, model_certification.html (Figure 3).

From this certification HTML file, the upper left frame, labeled "Certification Options", mimics the options from the "Certify Code" menu. For flexibility, the user can choose to create the rest of the stages for this safety policy, or the user can choose another safety policy and generate those stages. The browser window is separated into different frames.

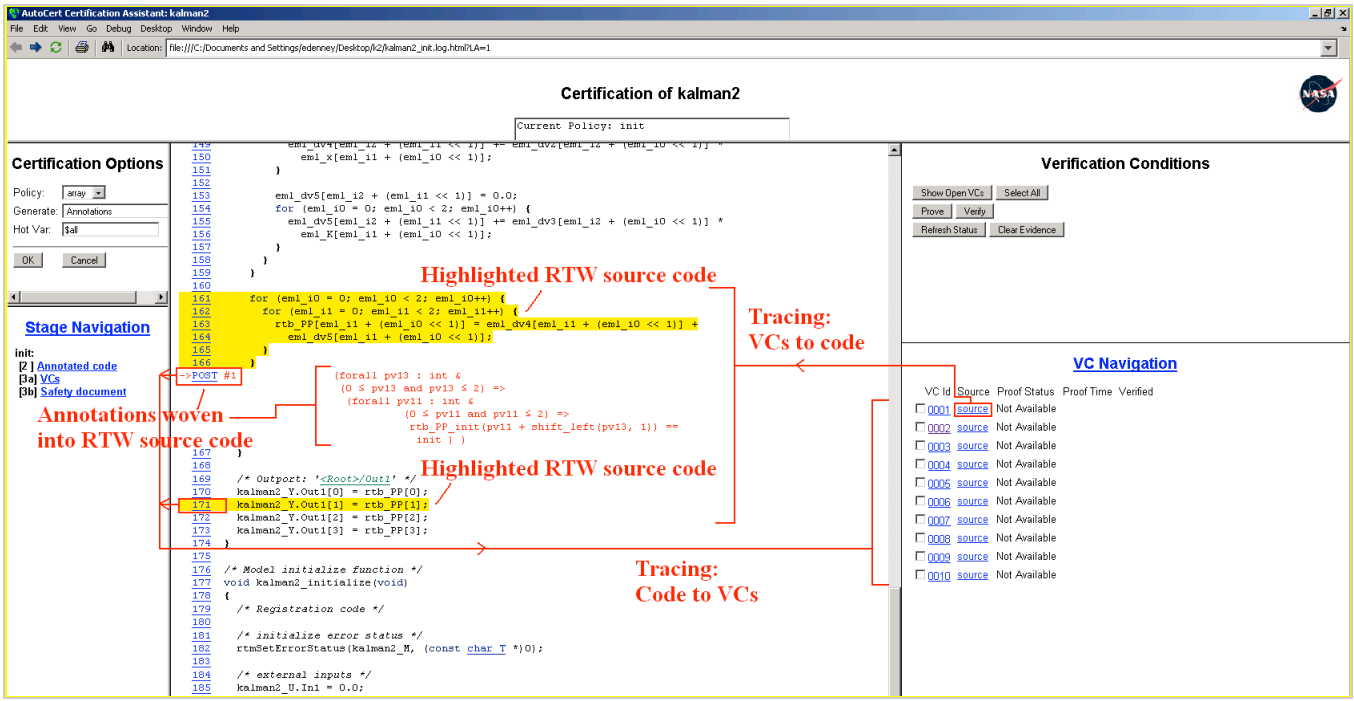

Figure 3 – AutoCert: Traceable Certification Results

Proving VCs and Verifying Proofs—The last phase of certificate creation is the proofs of user selected VCs using an Automated Theorem Prover (ATP). Also, to provide additional assurance, AutoCert allows the user to verify the proofs as well. This is an additional cross-check of the proofs generated by the theorem prover by an *independent* prover. This can be worthwhile if there is any concern over a local installation of a prover. Indeed, proofs can even be checked by entirely separate third-party tools. In Figure 3, the lower right hand frame (labeled VC Navigation) lists the VCs for the safety policy chosen by the user. The user can select any number of VCs from the list, and use the upper right hand frame (labeled Verification Conditions) to prove VCs or verify proofs of previously proven VCs. As VCs are proved or their proofs verified, the VC Navigation frame will update with extra information for the user to see, such as the "Proof Status" and "Proof Time", and whether the proof has been verified.

Tracing between Code and VCs—The user can browse the generated code, and by selecting a line, see the list of VCs (VC Navigation frame) that are dependent on that line (Figure 3).

The user can also select a VC and navigate to its source in the code. This action highlights the lines in the RTW-generated code (in the center pane of the browser) which "contribute" to the chosen VC (that is, they had either an annotation from which the VCG generated the given VC or contributed a safety obligation). Figure 3 shows how the tracing information can be used to support the certification process. A click on the source link associated with each VC prompts the certification assistant to highlight in yellow all affected lines of code, and annotations for the clicked VC are shown in the RTW-generated code (center frame). A click on the line number link at each line of code or a click on an annotation link will display all VCs associated with that line or annotation in the VC Navigation frame. In the VC Navigation frame, a further click on the verification condition link itself displays the formula which can then be interpreted in the context of the relevant program fragments. This helps domain experts assess whether the safety policy is actually violated, which parts of the program are affected, and eventually how any violation can be resolved. This traceability is also mandated by relevant standards such as DO-178B [RTC92].

In practice, safety checks are often carried out during code reviews, where reviewers look in detail at each line of the code and check the individual safety properties statement by statement. To support this, linking works in both directions: clicking on a statement or annotation displays all VCs to which it contributes. Along with RTW's model-to-code tracing capability, the code-to-VC tracing provides users with the ability to navigate from VCs to model elements. A more thorough and integrated functionality that permits a user to navigate directly from VCs to model and vice versa is planned.

Browsing and Navigation—When the certification HTML files are created, the user can choose which verification artifacts (i.e., certification stages) to view. The stage files are listed in the Stage Navigation frame (lower left hand frame in Figure 3). When the user clicks a link in that frame, the browser window displays the appropriate HTML files. This allows the user to create and view annotations, VCs, or certificates for different safety policies for the same

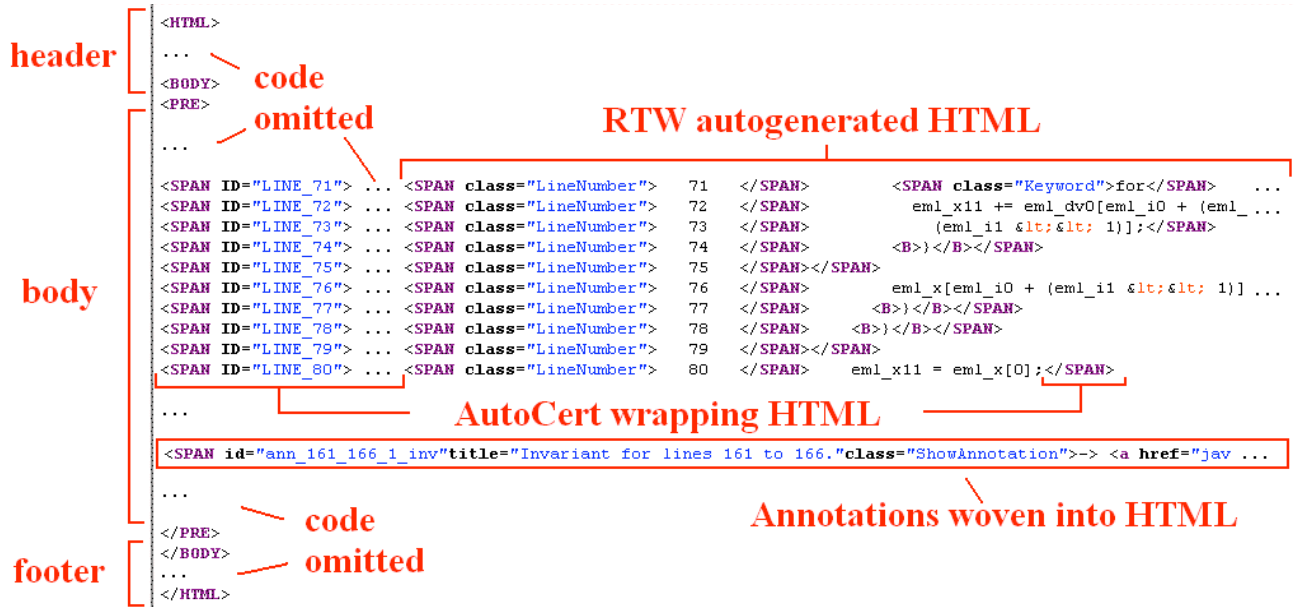

Figure 4 – AutoCert certification browser HTML source code

autogenerated RTW code without leaving the Matlab Browser window.

Implementation

We now consider the case of tracing code to VCs in some detail, as it is the one that requires the most additional functionality. There are two aspects to consider for implementing the interface. First, the mechanism by which tracing information can be incorporated into RTW-generated code; second, the representation format and language for implementing the tracing and controls (implemented as a backend to the inference engine).

Integration with RTW—There are a number of options for providing links from the code to the VCs. The first, and easiest, would be to generate parallel files that are similar in structure to their HTML but contain links to the VCs instead of links back to the model. However, this is not desirable from a usability standpoint as it would require the user to co-ordinate between two very similar files. This option was not considered further. A second approach would be possible if we had access to the HTML documentation templates used by RTW (similar to the way in which the code generation templates can be customized). In that case, we could alter the HTML generation so we can add in our own links (e.g., to the VCs) in addition to the ones to the model generated by EC. Navigating between code, VCs, and model would then be seamless. However, this would be contrary to the spirit of a plug-in which does not have access to generator internals. A third approach requires post-processing the generated HTML files to insert additional links. In addition, we can interweave additional information, such as annotations, into the generated HTML file. The new links at each line of code and each newly added annotation would bring traceability from code to VCs and vice-versa.

We chose option (3) as RTW does not currently provide access to the HTML code formatting templates, while option (2) would essentially require duplicating much of the work already done by the RTW/EC backend. For the weaver program, we implemented a parser specific to RTW HTML output. Because the RTW HTML file is well-formed, we are able to break the file into three parts: the header, the body, and the footer (Figure 4).

The header ends and the body begins at the HTML tag of `<PRE>`, and consequently the body ends and the footer begins at the HTML tag of `</PRE>`. A well-formed document conforms to all XML syntax rules. The main rule to understand is that every element with an opening tag is followed by a closing tag. Within the body, each source line in the HTML page represents an actual line of RTW code from the corresponding .c file. Once parsed, each of these source lines are wrapped with an HTML SPAN tag and given a unique HTML ID (the ID being the source line number).

The tool passes a list of inferred annotations from the autogenerated code, and the annotations are inserted into the correct locations in the combined RTW generated HTML with annotations and line numbers. The annotations are also wrapped with a SPAN tag and given a unique HTML ID (the ID being the unique annotation name). Using JavaScript, we explain how the user sees the tracing from code to VCs. With the HTML wrapping elements, we can highlight/unhighlight code and show/hide annotations accordingly. Further integration could be achieved if the files generated by AutoCert could be viewed in Model Explorer instead of the browser, but that would require modifying either the Matlab generated contents file or the template that generates it. At the time of writing, however, it appears that this is not possible, although MathWorks has indicated that it may be provided for in a future release.

AutoCert Backend—There are two alternatives for representing the tracing information and we discuss these now. One option is to retain as much as possible of the existing prototype, which is based on PHP. However, it would offer little possibility of integration into the Model Explorer component of RTW. This is because, unlike JavaScript, PHP must be executed on a web server. This in turn requires the user to switch between two different places: the browser for the VC traceability and Model Explorer for the rest of the functionality provided by RTW/EC. It also requires access to a PHP enabled web server. Instead, we chose a JavaScript-based approach.

JavaScript Based Implementation—A JavaScript based implementation allows us to integrate our functionality into Matlab in the most seamless manner. This is because JavaScript files are just HTML files with additional functions (defined in JavaScript) that are interpreted by the Matlab Browser. That is, they do not require an external web server. As mentioned above, since the Matlab Browser resides in the Matlab environment, it is possible to invoke Matlab command line calls from within the HTML files, giving access to AutoCert functionality.

For a JavaScript based implementation, there are a few commands needed:

- Make a system command line call from within an HTML file (JavaScript and Matlab command). This allows the AutoCert interface to:
    - Call the AutoCert inference engine
    - Call ATP systems to provide certificates
    - Call ATP systems to check certificates
- Highlight/unhighlight HTML code (JavaScript)
- Show/hide annotations (JavaScript)

Matlab allows the use of the `unix`, `system`, or '`!`' command to execute a program. For AutoCert, we used `system`. Using JavaScript we can invoke system calls using `system` and the `matlab:` protocol.

It is possible to call the software certification assistant tool and call ATP systems for generating and checking certificates. In the weaver program, we added HTML wrappers around each HTML element with a unique ID. This allows us to highlight/unhighlight source lines by changing the style color of the HTML element. Similarly, we can hide/show annotations by hiding/showing the HTML element wrapped around the annotations. The weaver program takes each line from the renumbered listing, except numbered references are replaced with their corresponding lines from the RTW-generated code, and weaves them together to generate calls to the predefined JavaScript functions.

Summary

We have described the integration of certification functionality (AutoCert) with the Matlab/RTW GUI in a way that preserves the user experience and is as seamless as possible. Existing RTW navigation is HTML based, so we have chosen to continue with that in order to preserve the user experience. A Matlab-based GUI approach was considered but rejected because it would not have been consistent with the HTML based approach used by Matlab.

4. CONCLUSIONS

The AutoCert system described here is a push-button verification technology for auto-generated code. The use of a tightly-coupled generation/analysis tool can allow system engineers to concentrate on the modeling and design, rather than worrying about low-level software details. By providing tracing and customizable safety reports, it supports both certification and debugging. We see AutoCert as a step towards providing an integrated "executive dashboard" for V&V.

The tool has two main benefits: it helps catch bugs in autocoders, and it helps with the certification process for the auto-generated code, thus mitigating the risk of using COTS autocoders that lack a trusted heritage.

Our approach offers a general framework for augmenting code generators with a certification component, and we have described an adaptation to MathWorks' Real-Time Workshop [MAT]. We have also developed a set of schemas adapted to a subset of the Simulink aerospace blockset [AERO]. Previous work concentrated on in-house code generators [4, 5].

The certification system based on annotation inference as described here is more flexible and extensible than decentralized architectures [GPCE05] where certification information is distributed throughout the code generator. Identifying patterns is an iterative process, but by allowing tracing between VCs and statements of the auto-generated code, the tool lets missing annotations and, thus, missing patterns, be pinpointed more easily.

By raising the level of abstraction at which verification knowledge is expressed, we are able to concisely capture many variations of the underlying code idioms. In particular, we can easily deal with optimizations which obscure low-level code structure. Indeed, there are other forms of guidance which are naturally expressed in a similarly declarative fashion, and we view annotation schemas as a first step towards a fully programmable certification language.

Finally, we are investigating other ways in which the analysis can provide insight into generated code. The safety report can form the basis of a *safety case*, that is, a top-down

argument for why the software meets its high-level requirements[3]. Another possibility is that by computing the weakest precondition of (the code generated by) a block/submodel, the tool can automatically determine its interface requirements. The user could also request that a specific submodel be certified (i.e., the code corresponding to that submodel).

REFERENCES

[DF03] Ewen Denney and Bernd Fischer. Correctness of source-level safety policies. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, Proc. FM 2003: Formal Methods}, volume 2805 of LNCS, pages 894-913, Pisa, Italy, September 2003. Springer.

[GPCE05] Ewen Denney and Bernd Fischer. Certifiable program generation. In Proceedings of the Conference on Generative Programming and Component Engineering (GPCE '05), volume 3676 of LNCS, pages 17-28, Tallinn, Estonia, September-October 2005. Springer.

[DFS06] Ewen Denney, Bernd Fischer, and Johann Schumann. An empirical evaluation of automated theorem provers in software certification. International Journal of AI Tools}, 15(1):81-107, February 2006.

[4] Ewen Denney and Bernd Fischer. Annotation inference for the safety certification of automatically generated code. In Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE '06), pages 265–268, Tokyo, Japan, September 2006. IEEE.

[5] Ewen Denney and Bernd Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In Proceedings of the Conference on Generative Programming and Component Engineering (GPCE '06), Portland, Oregon, October 2006. ACM Press.

[Erk04] Tom Erkkinen. Production code generation for safety-critical systems. Technical report, MathWorks, 2004.

[AERO] MathWorks Aerospace Blockset. http://www.mathworks.com/products/aeroblks/

[MAT] MathWorks Real-Time Workshop. http://www.mathworks.com/products/rtw

[RTC92] RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical report, RTCA, Inc., December 1992.

[SC03] Ingo Stürmer and Mirko Conrad. Test suite design for code generation tools. In Proceedings of 18th IEEE International Conference on Automated Software Engineering}, pages 286-290. IEEE, October 2003.

[SWC05] Ingo Stürmer, Daniela Weinberg, and Mirko Conrad. Overview of existing safeguarding techniques for automatically generated code. SIGSOFT Software Engineering Notes}, 30(4):1-6, July 2005.

[TPTP] Geoff Sutcliffe and Christian Suttner. TPTP home page. http://www.tptp.org.

BIOGRAPHY

Dr Ewen Denney *(PhD University of Edinburgh, 1999) has published over 40 papers in the areas of automated code generation, software modeling, software certification, and the foundations of computer science. He has been at NASA Ames for five years, where he has mainly worked on techniques for reliable automated code generation.*

Steven Trac *(University of Miami, 2008) is a research assistant in the Department of Computer Science at University of Miami, FL. His main research interest is in Computational Geometry. He is also a member of the Automated Reasoning Tools (ARTist) research group.*

[3] More precisely, a safety case is a structured argument that presents evidence for why a system remains safe in the presence of its known hazards. The first step, therefore, is a full hazard analysis